A Hardware Mechanism to Reduce the Cost of Forwarding Pointer Aliasing

J.P. Grossman, Jeremy Brown, Andrew Huang, Tom Knight

1 General Purpose

Forwarding pointers are an architectural mechanism that allow references to a memory location to be transparently forwarded to another location. Known variously as "invisible pointers" [Greenblatt74], "forwarding pointers" [Moon85] and "memory forwarding" [Luk99], they are familiar to the hardware community but to date have been incorporated into very few architectures.

One reason that forwarding pointers have received little support is that they have been perceived as possessing limited utility. Recently, however, it has become apparent that forwarding pointers are indeed useful constructs that can expedite program execution. In [Luk99] it is shown that using forwarding pointers to perform safe data relocation can result in significant performance gains on arbitrary programs written in C, speeding up some applications by more than a factor of two. In [Brown99] an algorithm is given for performing asynchronous local compacting garbage collection in a massively parallel distributed system. This algorithm uses forwarding pointers to avoid the high run-time costs usually associated with such a system. Thus, there is growing motivation to include hardware support for forwarding pointers in novel architectures.

A second and perhaps more significant reason that forwarding pointers have received little attention from hardware designers is that they introduce *aliasing*; it is possible for two different pointers to resolve to the same word in memory (Figure 1). The presence of this aliasing necessarily introduces run time costs in order to ensure correctness of execution. In [Luk99] two specific problems are identified. First, direct pointer comparisons are no longer a safe operation; some mechanism must be provided for determining the final addresses of the pointers. Second, seemingly independent memory operations may no longer be reordered in out-of-order machines.

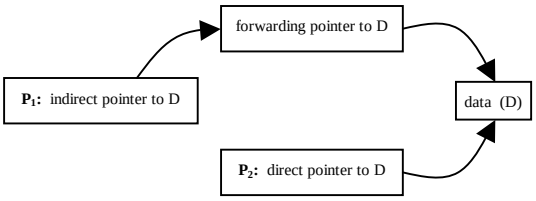

Figure 1: Forwarding pointer aliasing

This technology disclosure describes a simple mechanism which can be used to reduce the cost of forwarding pointer aliasing. Each object is assigned a short random tag, stored in pointers to the object, which is similar in role to a UID, but is not necessarily unique. Throughout this disclosure these tags will be referred to as **squids** (**S**hort **Q**uasi-**U**nique **ID**'s). In the common case squids allow pointer comparison and memory operation reordering to proceed with no overhead. Only in rare cases is it necessary to degrade performance to ensure correctness. Thus, squids allow an architecture to support forwarding pointers with reduced average run-time overhead. Furthermore, this overhead can be eliminated altogether if the software chooses not to make use of forwarding pointers.

2 Technical Description

The following discussion will assume the use of guarded pointers [Carter94]. Guarded pointers are a form of unforgeable capabilities [Fabry74] which include both a pointer and segment information within the guarded pointer itself. The specific format of the guarded pointer is not important, but we will assume that it is possible to determine the base address of an object given a pointer to the object's interior, as in [Carter94].

To implement squids, the guarded pointer format is augmented with a short n bit tag field. This tag field is randomly assigned each time an object is allocated. Unlike a UID, two pointers with the same squid may not point the same object. However, two pointers with different squids necessarily point to different objects. In many cases this fact can be used to avoid the run time costs of forwarding pointer aliasing.

Project Aries Technical Memo ARIES-TM-08
Artificial Intelligence Laboratory
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA

2.1 Pointer Comparisons

Two pointers can be efficiently compared by examining their base addresses, offsets and squids. If the base addresses are the same then the pointers point to the same object, and the pointers are the same if and only if they have the same offset into the object. If the squids are different then they point to different objects. It is only in the case that the base addresses are different but the squids and offsets are the same that it is necessary to perform expensive dereferencing operations to determine whether or not the final addresses are equal

It can be argued that this latter case will be rare. It occurs in two circumstances: either the pointers reference different objects which have the same squid, or the pointers reference the same object through different levels of indirection. The former of these occurs with probability 2^{-n}. The latter is application dependent, but we note that (1) applications tend to compare pointers to different objects more frequently than they compare pointers to the same object, and (2) the results of the simulations in [Luk99] indicate that it is reasonable to expect the majority of pointers to migrated data to be updated, so that two pointers to the same object will usually have the same level of indirection.

2.2 Reordering Memory Operations

In a similar manner, the hardware can decide whether or not it is possible to reorder memory operations based on squids. If two pointers have different offsets or different squids then the corresponding operations can be safely reordered. If the offsets and the squids are the same, then the operations are not reordered. No other mechanism is required to guarantee correctness of execution, and the probability of failing to reorder references to different objects is 2^{-n}.

2.3 Improving Performance

The use of squids reduces the average overhead necessary to check for aliasing to a small but still non-zero amount. Ideally, pointers to objects which are never migrated should incur no overhead whatsoever. This can be achieved by adding a single 'migrated' bit (M) to the guarded pointer format which indicates whether or not the pointer points to the original address at which the object was allocated. When a new object is created, pointers to that object have M = 0. When the object is migrated, pointers to the new location (and all subsequent locations) have M = 1. If the hardware is comparing two pointers with M = 0 (either as the result of a user comparison instruction, or to determine whether or not memory operations can be reordered), it can ignore the squids and perform the comparison based on the addresses alone. Hence, there is no runtime cost associated with support for forwarding pointers until the software makes use of them.

2.4 Hardware and Software Overhead

The only software overhead required to support squids is the code that generates them when objects are allocated, which adds a few instructions to memory allocation. A trap handler is needed to check for aliasing when comparing different addresses with the same squid, but this code (or an equivalent hardware mechanism) is a general requirement for supporting forwarding pointers without UIDs and is not specific to the implementation of squids. Moreover, placing a single copy of this code in a trap handler creates much less software overhead than inlining the code at every pointer comparison as in [Luk99].

In order to be effective, squids require only a small number of bits to be added to the guarded pointer format. For example, if eight bits are added (seven squid bits and one migrated bit, as in Figure 2), then the probability of failing to distinguish pointers to different objects is less than 0.008. In addition to an augmented guarded pointer format, the hardware required to implement squids consists of some simple logic to inspect squid/M bits for pointer comparisons and memory operation reordering, and support for a trap which occurs when different pointers with the same squids are compared.

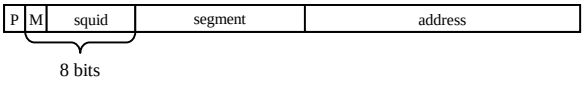

Figure 2: A guarded pointer consists of an address, segment information, and a single pointer bit P to distinguish guarded pointers from data. To support squids, a single migrated bit M and a small number of squid bits (seven as shown) are added.

3 Advantages over Existing Methods

In [Luk99] the problem of pointer comparisons is addressed by inserting code to determine the final address for each pointer (unless the compiler can somehow determine that the pointers do not point to relocated objects). The overhead of this approach is potentially large. In the best case, both target memory words will be resident in the cache, neither of them will contain a forwarding pointer, and the

pointer comparison will be slowed down by roughly an order of magnitude. However, since pointer comparisons often precede a decision to perform operations on an object, a common case will be for one or both dereferences to cause a cache miss, slowing down the comparison by another order of magnitude. In the worst case, one of the pointers points to data which is not even resident in main memory, which can occur frequently in programs that deal with massive datasets. It is desirable to be able to compare pointers without having to dereference them; by using squids it is possible to do so in the majority of cases.

The solution proposed in [Luk99] for reordering memory operations is to use *data dependence speculation*, which allows loads to execute speculatively before it is known that they are independent of any preceding stores. In an architecture that supports data dependence speculation, it is fairly easy to extend the hardware to operate correctly in the presence of forwarding pointers. In [Luk99] it was found that this solution is effective as incorrect speculation occurs only rarely. However, it assumes the presence of some fairly complex hardware. For architectures in which silicon area efficiency is a concern, a lower cost alternative such as squids is preferable.

The forwarding pointer aliasing problem is an instance of the more general challenge of determining object identity in the presence of multiple and/or changing names. This problem has been studied explicitly [Setrag86]. A natural solution which has appeared time and again is the use of system-wide unique object ID's (e.g. [Dally85], [Setrag86], [Moss90], [Day93], [Plainfossé95]). UID's completely solve the aliasing problem, but have two disadvantages:

i. The use of ID's to reference objects requires an expensive translation each time an object is referenced to obtain the virtual address of the object.

ii. Quite a few bits are required to ensure that there are enough ID's for all objects and that globally unique ID's can be easily generated in a distributed computing environment. In a large system, at least sixty-four bits would likely be required in order to avoid any expensive garbage collection of ID's and to allow each processor to allocate ID's independently.

Squids maintain one of the advantages of UID's, namely they allow pointers to different objects to be distinguished quickly (with high probability). They do so with only a small number of bits, and without requiring any translation tables (since they are a part of the guarded pointer format).

4 Commercial Applications

Forwarding pointers are a key enabling mechanism for safe data compaction and efficient garbage collection. The mechanisms described in this disclosure allow forwarding pointer support to be incorporated into novel architectures with little or no average run-time cost due to aliasing.

In [Luk99] the advantages of data relocation in the context of a uniprocessor were made clear. By compacting live data, better use can be made of the cache and as a result program execution is sped up by as much as a factor of two. In a distributed shared memory multiprocessor it is also important to be able to relocate data for a different reason: a processor can access local memory an order of magnitude faster than it can access remote memory. Effective computation on such a machine therefore depends on being able to move data to the processing node at which it is needed. Squids provide efficient support for data migration and are therefore applicable to both single processor and multiprocessor high performance systems.

Historically, one of the primary uses of forwarding pointers has been to implement incremental garbage collection ([Baker78], [Moon84]). More recently, it is shown in [Brown99] that forwarding pointers can be used to implement efficient local compacting garbage collection in a massively parallel distributed system. Hardware support for fast garbage collection is especially important given the growing prevalence of the Java programming environment, which is the language of choice for web programming and which specifies a garbage collected memory model [Gosling96]. Another application of squids is therefore the implementation and/or improvement of systems which are specifically designed to run Java efficiently (such systems are already under development, e.g. [Tremblay99]).

References

[Baker78] Henry G. Baker, Jr., "List Processing in Real Time on a Serial Computer", Communications of the ACM, Volume 21, Number 4, pp. 280-294, April 1978.

[Brown99] Jeremy Brown, "Memory Management on a Massively Parallel Capability Architecture",

Ph.D. thesis proposal, M.I.T., December 1999.

[Carter94] Nicholas P. Carter, Stephen W. Keckler, William J. Dally, "Hardware Support for Fast Capability-based Addressing", Proc. 6th International Conference on Architectural Support for Programming Languages and Operating Systems, 1994.

[Dally85] William J. Dally, James T. Kajiya, "An Object Oriented Architecture", Proc. ISCA '85, pp. 154-161.

[Day93] Mark Day, Barbara Liskov, Umesh Maheshwari, Andrew C. Myers, "References to Remote Mobile Objects in Thor", ACM Letters on Programming Languages & Systems, vol.2, no.1-4, March-Dec. 1993, pp.115-26.

[Fabry74] R.S. Fabry, "Capability-Based Addressing", Communications of the ACM, Volume 17, Number 7, pp. 403-412, July 1974.

[Gosling96] James Gosling, Bill Joy, Guy L. Steele Jr., The Java Language Specification, Addison-Wesley Publication Co., Sept. 1996, 825pp.

[Greenblatt74] Richard Greenblatt, "The LISP Machine", Working Paper 79, M.I.T. Artificial Intelligence Laboratory, November 1974.

[Luk99] Chi-Keung Luk, Todd C. Mowry, "Memory Forwarding: Enabling Aggressive Layout Optimizations by Guaranteeing the Safety of Data Relocation", Proc. ISCA '99, pp. 88-99.

[Moon84] David A. Moon, "Garbage Collection in a Large Lisp System", Proc. 1984 ACM Conference on Lisp and Functional Programming, pp. 235-246.

[Moon85] David A. Moon, "Architecture of the Symbolics 3600", Proc. ISCA '85, pp. 76-83, 1985.

[Moss90] J. Eliot B. Moss, "Design of the Mneme Persistent Object Store", ACM Transactions on Information Systems, Vol. 8, No. 2, April 1990, pp. 103-139.

[Plainfossé95] David Plainfossé, Marc Shapiro, "A Survey of Distributed Garbage Collection Techniques", Proc. 1995 International Workshop on Memory Management, pp. 211-249.

[Setrag86] Setrag N. Khoshafian, George P. Copeland, "Object Identity", Proc. 1986 ACM Conference on Object Oriented Programming Systems, Languages and Applications, pp. 406-416.

[Tremblay99] Marc Tremblay, "An Architecture for the New Millenium", Proc. Hot Chips XI, Aug. 15-17, 1999.